CENTRUL DE PREGĂTIRE PENTRU PERFORMANŢĂ HAI LA OLIMPIADĂ !

DISCIPLINA INFORMATICĂ

JUDEŢUL SUCEAVA

Titlul lectiei: Divide et impera

Data: 29.10.2016

Profesor: Ilincăi Florin

Grupa: clasa a X-a locul de desfasurare: CN PR Suceava

Metoda Divide et impera

Această metodă este utilizată în rezolvarea unor probleme complexe, care au următoarele caracteristici:

- Pot fi descompuse în probleme de complexitate mai mică. Acestea sunt fie de acelaşi tip cu problema inițială, fie au soluție imediată (probleme primitive);
- Rezolvarea problemelor rezultate în urma descompunerii este mai uşoară decât rezolvarea problemei inițiale;
- Prin combinarea soluțiilor problemelor rezultate în urma descompunerii se obține o soluție pentru problema inițială.

Pentru fiecare din problemele rezultate în urma descompunerii se aplică acelaşi procedeu de descompunere. Algoritmul este recursiv, implementarea fiind realizată de obicei prin subprograme recursive.

rezolvarea unei astfel de probleme solicită trei faze:

Divide problema într-un număr de subprobleme independente, similare problemei inițiale, de dimensiuni din ce în ce mai mici; descompunerea subproblemelor se opreşte la atingerea unor dimensiuni care permit o rezolvare simplă;

Stăpâneşte subproblemele prin rezolvarea acestora în mod recursiv. Dacă dimensiunile acestora sînt suficient de mici, rezolvă problemele în mod uzual, nerecursiv;

Combină soluțiile tuturor subproblemelor în soluția finală pentru problema inițială.

Modelul matematic:

P(n): problema de dimensiune n

baza

daca $n< n_0$ atunci rezolva P prin metode elementare

divide-et-impera

- divide P in a probleme $P_1(n_1),...,P_a(n_a)$ cu $n_a < n/b$, $b>1$
- rezolva $P_1(n_1)$, ..., $P_a(n_a)$ in aceeaşi maniera şi obține soluțiile $S_1, ..., S_a$

● asamblează S_1, ..,S_a pentru a obține soluția S a problemei P

Pentru aceşti algoritmi există atît implementarea recursivă cît şi iterativă, cea mai flexibilă şi cea mai utilizată fiind varianta recursivă.

```
procedure DivideEtImperaR(P, n, S)
    if (n <= n0) then
        determina S prin metode elementare
        else
        imparte P in P1, ..., Pa
    DivideEtImperaR(P1, S1)
        .....
    DivideEtImperaR(Pa, Sa)
    Asambleaza(S1, ..., Sa, S)
```

Algoritmii divide et impera au o bună comportare în timp, dacă dimensiunile subproblemelor în care se împarte subproblema sînt aproximativ egale. Dacă lipsesc fazele de combinare a soluțiilor, viteza acestor algoritmi este şi mai mare (ex. căutarea binară). În majoritatea cazurilor, descompunerile înjumătățesc dimensiunea problemei, un exemplu tipic reprezentându-l sortarea prin interclasare.

Maximul dintr-un şir de numere

Să presupunem că se doreşte aflarea valorii maxime dintr-o secvenţă de n numere: {a_1, a_2, ..., a_n}. Una din metodele de rezolvare constă în determinarea valorii cea mai mare din prima jumătate (fie aceasta x_1), apoi se determină valoarea cea mai mare din a doua jumătate (fie aceasta x_2). Soluţia problemei iniţiale este maximul dintre cele două valori găsite max(x_1, x_2). Problema iniţială a fost descompusă în două subprobleme de acelaşi tip, dar mai simple deoarece lungimea fiecărei secvenţe este jumătate din lungimea secvenţei iniţiale.

Nu este obligatorie descompunerea problemei iniţiale în două jumătăţi. Problema poate fi rezolvată pe baza unei metode care o reduce succesiv la o problemă mai simplă, determinându-se valoarea maximă din primele n-1 componente ale secvenţei (fie aceasta x_1), soluţia problemei fiind max (x_1, a_n).

Alegerea modului de descompunere se face în funcţie de problema rezolvată şi de experienţa programatorului. Pentru ambele opţiuni descompunerea continuă până se ajunge la probleme cu soluţie imediată (determinarea maximului dintr-o secvenţă cu un singur element).

Soluție:

divide-et-impera

- divide: m = [(p + q)/2]
- subprobleme: maxim(a[p..m]), maxim(a[m+1..q])
- asamblare: se combină soluțiile subproblemelor maxim(a[p..m]) şi maxim(a[m+1..q])

```
#include <iostream>
#include <fstream>

using namespace std;
int v[100],n;
ifstream f("numere.in");
```

```
int maxim(int li, int ls)
{
   int m,a,b;
   if(li==ls)
      return v[li];
   else
   {
      m=(li+ls)/2;
      a=maxim(li,m);
      b=maxim(m+1,ls);
      if(a>b)
         return a;
      else
         return b;
   }
}

int main()
{
   f>>n;
   for(int i=1;i<=n;i++)
      f>>v[i];
   cout << "max=" << maxim(1,n);
   return 0;
}
```

Turnurile din Hanoi

Fie 3 tije şi n discuri cu diametre distincte. Notăm tijele cu A,B,C. Cele n discuri se află toate pe tija A, aşezate în ordine descrescătoare a diametrelor. Se cere să se mute cele n discuri de pe tija A pe B, folosind drept tijă de manevră tija C, astfel:
- la un moment dat se mtă un singur disc;
- niciodată un disc de diametru mai mare nu se află peste un disc de diametru mai mic.

Rezolvarea problemei constă în mutarea a n-1 discuri de pa A pe C folosid ca manevră tija B, apoi mutarea discului n de pe A pe B, apoi se aduc cele n-1 discuri de pe C pe B folosind ca manevră tija A.

Soluție:

divide-et-impera

subprobleme: *hanoi(n-1,a,c,b),hanoi(1),hanoi(n-1,c,b,a);*

asamblare: *nu există, deoarece soluția problemei inițiale a fost obținută simultan cu etapele de descompunere*

```
#include <iostream>
using namespace std;

void hanoi(int n, char a, char b, char c)
{
   if(n==1)
      cout<<a<<"->"<<b<<endl;
   else
   {
      hanoi(n-1,a,c,b);
      cout<<a<<"->"<<b<<endl;
```

```
    hanoi(n-1,c,b,a);
  }
}

int main()
{
  int n;
  cout<<"n="; cin>>n;
  hanoi(n,'A','B','C');
  return 0;
}
```

CMMDC a n numere naturale

Şirul de numere pentru care se doreşte determinarea cmmdc este memorat într-n vector. Vectorul va fi împărţit în două, apoi fiecare din acestea va fi împărţită în alte două părţi mai mici, până când vectorii astfel obţinuţivor avea cel mult două elemente. Pentru aceste elemente aplicăm algoritmul lui Euclid de determinare a cmmdc.

```
#include <iostream>
#include<fstream>

using namespace std;
int v[100], n;
ifstream f("intrare.in");
int euclid(int a, int b)
{
  int r;
  while(b)
  {
    r=a%b;
    a=b;
    b=r;
  }
  return a;
}

int cmmdc(int li, int ls)
{
  int m;
  if(ls-li<=1)
    return euclid(v[li],v[ls]);
  m=(li+ls)/2;
  return euclid(cmmdc(li,m), cmmdc(m+1, ls));
}
int main()
{
  int i;
  f>>n;
  for(i=1;i<=n;i++)
    f>>v[i];
  cout << "cmmdc=" <<cmmdc(1,n);
  return 0;
}
```

Algoritmi de sortare

Sortarea prin interclasare (Merge sort)

Se consideră vectorul a cu n componente numere întregi (sau reale). Să se sorteze crescător utilizând mecanismul sortării prin interclasare.

Algoritmul interclasării a doi vectori este performant deoarece se efectuează cel mult m+n-1 comparaţii.

Algoritmul sortării prin interclasare se bazează pe următoarea idee: împărţirea vectorului în doi vectori care, odată sortaţi se interclasează. Conform strategiei generale Divide et impera, problema este descompusă în alte două subprobleme de acelaţi tip şi, după rezolvarea lor, rezultatele se combină (interclasează). Descompunerea vecorului în alţi doi vectori care urmează a fi sortaţi are loc până când avem de sortat vectori de una sau două componente.

Funcţia sortare sortează un vector de maxim două componente.

Funcţia intercl interclasează rezultatele.

Funcţia divimp implementează strategia generală a metodei studiate.

Schema problemei:

- generalizare: *sortează(a[p..q])*
- baza: p ≥q

Soluție:

divide-et-impera

- divide: m = [(p + q)/2]
- subprobleme:*sortează(a[p..m]), sortează(a[m+1..q])*
- asamblare: interclaseaza subsecvențele sortate a[p..m] şi a[m+1..q]

complexitate: T(n) = O(n log n)

```
#include <iostream>
#include <fstream>

using namespace std;

int a[100],n;
fstream f("numere.in");

void sortare(int p, int q, int a[100])
{
   int m;
   if(a[p]>a[q])
   {
      m=a[p];
      a[p]=a[q];
      a[q]=m;
   }
}
void intercl(int p, int q, int m, int a[10])
{
   int b[100], i,j,k;
   i=p;
   j=m+1;
   k=1;
   while(i<=m&&j<=q)
```

```
        if(a[i]<=a[j])
    {
        b[k]=a[i];
        i++; k++;
    }
    else
        {
         b[k]=a[j];
          j++; k++;
        }
        if(i<=m)
            for(j=i;j<=m;j++)
            {
                b[k]=a[j];
                k++;
            }
        else
        for(i=j;i<=q;i++)
            {
                b[k]=a[i];
                k++;
            }
            k=1;
            for(i=p;i<=q;i++)
            {
                a[i]=b[k];
                k++;
            }

}
void divimp(int p, int q, int a[20])
{
    int m;
    if((q-p)<=1)
        sortare(p,q,a);
    else
    {
        m=(p+q)/2;
        divimp(p,m,a);
        divimp(m+1,q,a);
        intercl(p,q,m,a);
    }
}
int main()
{
    int i;
    f>>n;
    for(i=1;i<=n;i++)
       f>>a[i];
    divimp(1,n,a);
     for(i=1;i<=n;i++)
       cout<<a[i]<<" ";
```

```
    return 0;
}
```

Sortarea rapidă (Quick sort)

Pentru implementarea metodei vom folosi o funcţie poz care tratează o porţiune din vector cuprinsă între indicii li şi ls. Rolul acestei funcţii este de a poziţiona prima componentă a[li] pe o poziţie k cuprinsă între li şi ls astfel încât toate componentele vectorului cuprinse între li şi k-1 sa fie mai mici decât a[k] şi toate componentele vectorului cuprinse între k+1 şi ls să fie mai mari sau egale decât a[k].

Această funcţie există două moduri de lucru:

a. i rămâne constant, j scade cu 1;
b. i creşte cu 1, j rămâne constant.

Funcţia este concepută astfel;

- Iniţial i va lua valoarea li iar j valoarea ls (elementul care iniţial se află pe poziţia li se va găsi mereu pe o poziţie dată de i sau de j).
- Se trece în modul de lucru a
- Atât timp cât i<j, se execută:
  - Dacă a[i] este strict mai mare decât a[j] atunci se inversează cele două numere şi se schimbă modul de lucru
- i şi j se modifică corespunzător modului de lucru în care se află programul
- k ia valoarea comună a lui i şi j;

Exemplu:
Fie a=(6,9,3,1,2).

- li=1, ls=5, modul de lucru a).
  - i=1, j=5; a[1]>a[5], se inversează elementele aflate pe poziţiile 1 şi 5 deci a=(2,9,3,1,6) şi se trece în modul de lucru b).
  - i=2, j=5; a[2]>a[5], se inversează elementele aflate pe poziţiile 2 şi 5 deci a=(2,6,3,1,9) şi se trece în modul de lucru a).
  - i=2, j=4; a[2]>a[4], se inversează elementele aflate pe poziţiile 2 şi 4 deci a=(2,1,3,6,9) si se trece în modul de lucru b).
  - i=3, j=4
  - i=4, j=4
- funcţia se încheie, elementul aflat iniţial pe poziţia 1 se găseşte acum pe poziţia 4, toate elementele din stânga lui sunt mai mici decât el, toate elementele aflate în dreapta lui fiind mai mari decât el (k=4).

Alternanţa modurilor de lucru se explică prin faptul că elementul care trebuie poziţionat se compară cu un element care se află în dreapta sau în stânga lui, ceea ce impune o modificare corespunzătoare a indicilor i şi j.

După aplicarea funcţiei poz, elementul aflat iniţial pe poziţia li va ajunge pe o poziţie k şi va rămâne pe acea poziţiei în cadrul vectorului deja sortat.

Funcţia quick are parametrii li şi ls. În cadrul ei se utilizează metoda Divide et impera astfel:

- se apelează poz;
- se apelează Quick pentru li şi k-1;
- se apelează Quick pentru k+1 şi ls.

```
#include <iostream>
#include <fstream>

using namespace std;

int a[100],n,k;
fstream f("numere.in");

void poz(int li, int ls, int & k, int a[100])
{
   int i=li, j=ls, c, i1=0,j1=-1;
   while(i<j)
   {
      if(a[i]>a[j])
      {
         c=a[i];
         a[i]=a[j];
         a[j]=c;
         c=i1;
         i1=-j1;
         j1=-c;
      }
   i+=i1;
   j+=j1;
   }
   k=i;
}
void quick(int li, int ls)
{
     if(li<ls)
    {
       poz(li,ls,k,a);
       quick(li,k-1);
       quick(k+1,ls);
   }
}
int main()
{
   int i;
   f>>n;
   for(i=1;i<=n;i++)
      f>>a[i];
  quick(1,n);
    for(i=1;i<=n;i++)
      cout<<a[i]<<" ";

   return 0;
}
```

Problema tăieturilor

O placă dreptunghiulară de dimensiuni l şi h conține n găuri de dimensiuni punctiforme. Cunoscând coordonatele fiecărei găuri să se determine aria maxima a unei suprafeţe fără găuri care se poate obţine din placa dată efectuând tăieturi paralele cu laturile plăcii.

Schema problemei:

- generalizare: xj,yj,xs,ys (coordonatele dreptunghuilui), n, x[n] ,y [n] , dreptunghi[i,xj,yj,xs,ys]

Soluție:

- divide-et-impera
  - subprobleme:
    dreptunghi [i+1,xj,yj,xs,y[i]],
    dreptunghi [i+1,xj,y i ,xs,ys],
    dreptunghi [i+1,xj,yj,x i s,ys],
    dreptunghi [i+1, x i ,yj,xs,ys].
  - asamblare: nu există.

Dacă toate punctele sunt exterioare dreptunghiului dat sau se află pe laturile sale, dreptunghiul căutat este chiar cel inițial.

Dacă există un singur punct interior dreptunghiului, ducând prin punctul respectiv o paralelă la una din axele de coordonate se obțin alte două dreptunghiuri care nu mai conțin punctul interior.

Din cele patru dreptunghiuri formate va trebui reținut cel de arie maximă. Dacă există două puncte interioare dreptunghiului, al doilea punct ar putea fi punct interior la doar unul dintre dreptunghiurile D1, D2, respectiv D3, D4, regăsim astfel problema inițială pentru dreptunghiurile D1, D2, D3, D4 şi punctul al doilea, toate acestea conducând la o abordare
divide et impera. Problema inițială se descompune în patru subprobleme corespunzătoare celor patru dreptunghiuri formate prin trasarea celor două paralele prin primul punct interior al dreptunghiului. Fie acesta punctul i de coordonate (xi,yi). În continuare, punctele interioare vor putea fi numai punctele j i+1,.., n . Descompunerea continuă în acelaşi mod, până se ajunge la dreptunghiuri fără puncte interioare ale căror arii se calculează şi se reține maximul.

```
#include <iostream>
#include <fstream>

using namespace std;
struct punct {
   int x,y;
};

int X, Y;
punct v[20];
int n;
int S;
int xs, ys, xd, yd;

void taiere(int x1, int y1, int x2, int y2) {
   //x1,y1; x2,y2 - coordonatele coltului stanga-jos,dreapta-sus;
   int g=0, poz;
   for(int i=1; i<=n && !g; i++) //ma opresc cand dau peste o perforatie in interiorul suprafetei
      if(v[i].x>x1 && v[i].x<x2 && v[i].y>y1 && v[i].y<y2) {
         g=1;
         poz=i;
      }
```

```
    if (g==1) {
       taiere(x1, v[poz].y, x2, y2); // deasupra
       taiere(x1, y1, x2, v[poz].y); // dedesubt
       taiere(x1, y1, v[poz].x, y2); // stanga
       taiere(v[poz].x, y1, x2, y2); // dreapta
      }
    else if ((x2-x1)*(y2-y1)>S) {
       xs=x1;
       ys=y1;
       xd=x2;
       yd=y2;
       S=(x2-x1)*(y2-y1);
    }
}
int main()
{
    ifstream f("placa.in");
    f>>n;
    for(int i=1; i<=n; i++)
       f>>v[i].x>>v[i].y;
    f>>X>>Y;
    taiere(0, 0, X, Y);
    cout<<endl<<"Cea mai mare suprafata neperforata are coordonatele
("<<xs<<","<<ys<<");("<<xd<<","<<yd<<")"<<endl;
    cout<<"suprafata= "<<S;
    return 0;
}
```

Tema:

1. **Enunț:** Scrieți un algoritm divide et impera care să calculeze valoarea unui polinom într-un punct.

Schema problemei:
˘ generalizare: a[p..q]
˘ baza: p≥q

Soluție:
˘ divide-et-impera
divide: m = [n/2]
subprobleme: a[p..m], a[m+1..q]
asamblare: determină valoarea finală prin combinare rezultatelor subsecvențelor
Fie polinomul $P(x)=a_nx^n+..+a_1x+a_0$. Vom calcula valoarea sa în punctul y folosind exprimarea $P(y)=Q(y)* y^{[n/2]} +T(y)$, urmând să evaluăm polinoamele Q şi T în x. Pentru Q şi T folosim o scriere analoagă, până când gradul lor este 0 sau 1.

2. **Enunț:** Se consideră un vector de lungime n. Definim plierea vectorului prin suprapunerea unei jumătăți, numită donatoare, peste cealaltă jumătate, numită receptoare, cu precizarea că dacă numărul de elemente este impar, elementul din mijloc este eliminat. Prin pliere, elementele subvectorului obținut vor avea numerotarea jumătății receptoare. Plierea se poate aplica în mod repetat, până când se ajunge la un subvector format dintr-un singur element, numit element final. Scrieți un program care

să afişeze toate elementele finale posibile şi să se figureze pe ecran pentru fiecare element final o succesiune de plieri.

Schema problemei:
ˇ generalizare: pliază[p..q]
ˇ baza: p≥q
Soluție:
divide-et-impera
divide: m = [n/2]
subprobleme:
dacă (q-p+1) mod 2=1 *atunci* Ls=(p+q)div 2-1 *altfel* Ls=(p+q)div 2; Ld=(p+q) div 2 +1;
pliază[p..Ls], *pliază*[Ld..q]
asamblare: nu există;

Pentru determinarea tuturor elementelor finale şi succesiunile de plieri corespunzătoare pentru un vector cu numerotarea elementelor p..q, vom utiliza o procedură Pliaza(p,q). Dacă p=q, atunci vectorul este format dintr-un singur element, deci final. Dacă p<q, calculăm numărul de elemente din vector (q-p+1).

Dacă numărul de elemente din vector este impar, elementul din mijloc ((p+q)div2) este eleminat, plierea la stânga se face de la poziția (p+q)div2 -1, iar plierea la dreapta de la poziția (p+q) div2+1.

Dacă numărul de elemente din vector este par, plierea la stânga se poate face de la poziția (p+q) div 2 iar plierea la dreapta de la poziția (p+q)div2+1. Vom codifica plierea la stînga reținând şirul mutărilor simbolul 'S' urmat de poziția Ls, de la care se face plierea spre stânga, iar o pliere la dreapta prin simbolul 'D', urmat de poziția Ld de la care se face plierea spre dreapta. Pentru a determina toate elementele finale din intervalul p..q, apelăm recursiv procedura Pliaza pentru prima jumătate a intervalului, precedând toate succesiunile de plieri cu o mutare spre stânga, apoi apelăm procedura Pliaza pentru cea de-a doua jumătate a intervalului, precedând toate succesiunile de plieri corespunzătoare de o pliere la dreapta. Exemplu, pentru n=7, elementele finale şi plierile corespunzătoare sînt:
1: S3 S1
3: S3 D3
5: D5 S5
7: D5 D7